Lecture #10: Dynamic Analysis for Security

UCalgary ENSF619

Elements of Software Security

Instructor: Lorenzo De Carli (lorenzo.decarli@ucalgary.ca)

Partly based on slides by Drew Davidson, University of Kansas, and

"FuzzGen: Automatic Fuzzer Generation", Ispoglou et al., USENIX Sec. 2020

Today’s lecture is about dynamic analysis

- Well, in part, because today’s paper proposes dynamic analysis informed by static analysis
  - It’d be too easy otherwise
- Contrarily to static analysis, **dynamic analysis** monitors a program **while it is running** to **perform measurements** or **infer facts**

Forms of dynamic analysis

- Profiling/measurement
  - Typically instrument a program to generate telemetry
  - Example: measuring code coverage, tracing system calls
- Bug detection
  - Observe program functioning to detect bugs
  - Oftentimes performed together with instrumentation
  - Example: fuzzing

Advantages of dynamic analysis

- Measures a **real-world execution** of a program
- It is hard to argue with its results, as it is based on events (program traces) that **actually happened**
- Contrast with static analysis, which can be **imprecise** in the presence of certain constructs
- It is typically **easier to deploy** than static analysis as there is plenty of tooling for instrumentation/controlled execution
- **Performance is less of a concern** (executing the program VS analyzing a complex model of the program)

Disadvantages of static analysis

- **Impossible to generalize**
- The fact that, for example, a bug was not observed in one execution does not mean that the program is bug-free
- A dynamic analysis algorithm is typically:
  - **Sound** (if it detects a problem, there is a problem)
  - But **not complete** (if it does not detect a problem, a problem may still exist)
- For example, the fact that a fuzzer does not trigger any **bug does not mean** that a program is bug-free

Review of a few basic concepts

Code coverage analysis

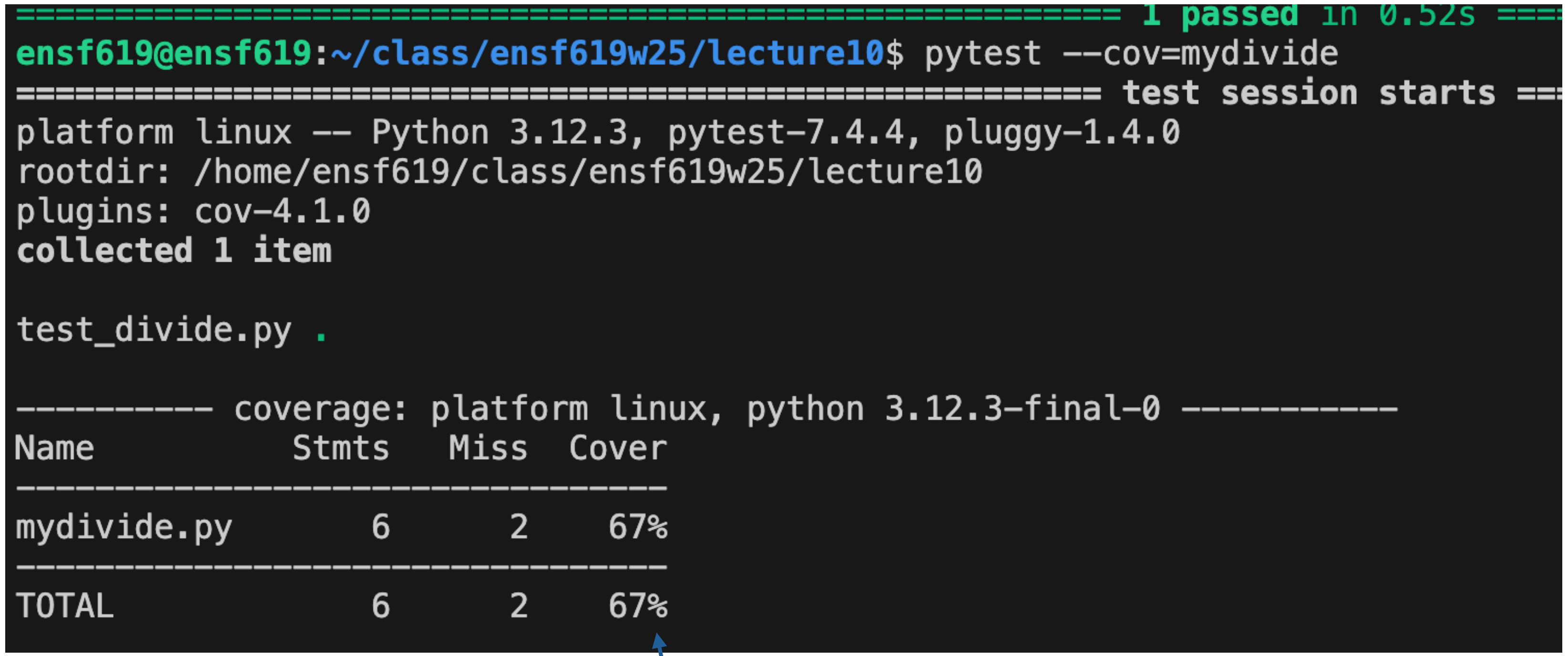
- When running tests against a program, a reasonable question is: **how much of the code** is executed by those **tests**?
- **Why is this an important question?**

Toy code coverage example

```python
def divide(a, b, is_int=True):
  if is_int:
    if b == 0:
      return None
    return a // b
  else:
    return a / b

def test_divide():
  assert divide(4, 0) == None
```

- **How many lines** are exercised by this test?
- Is this a good test suite?
- **…why?**

Let's see a demo



```
=========================================================== 1 passed in 0.52s ====
ensf619@ensf619:~/class/ensf619w25/lecture10$ pytest --cov=mydivide
======================================================= test session starts ===
platform linux -- Python 3.12.3, pytest-7.4.4, pluggy-1.4.0
rootdir: /home/ensf619/class/ensf619w25/lecture10
plugins: cov-4.1.0
collected 1 item

test_divide.py .

---------- coverage: platform linux, python 3.12.3-final-0 -----------
Name           Stmts   Miss  Cover
----------------------------------
mydivide.py        6      2    67%
----------------------------------
TOTAL              6      2    67%
```

Tests only gives information about the portion of the code that was covered

Fuzzing

- **Fuzzing** is a form of **dynamic analysis** dedicated to **finding bugs** (and oftentimes **vulnerabilities**)
- Idea: run a program with **randomized inputs**, until it **crashes** or a maximum **time limit** is **exceeded**
- On the face of it, it sounds like a silly idea, but it can be made **very useful**
- Fuzzing is independent from code coverage, although oftentimes **relies on it**

Fuzzing guided by code coverage

- Most (reasonably written) programs will perform **sanity checks** on **inputs**
- Submitting a **large number** of **random inputs** will most likely cause the program to **reject all of them**, thus causing **very limited code coverage**
- Thus, fuzzing by blindly **submitting random inputs** is **not very helpful**
- **Code coverage** can provide guidance to optimize fuzzing efforts

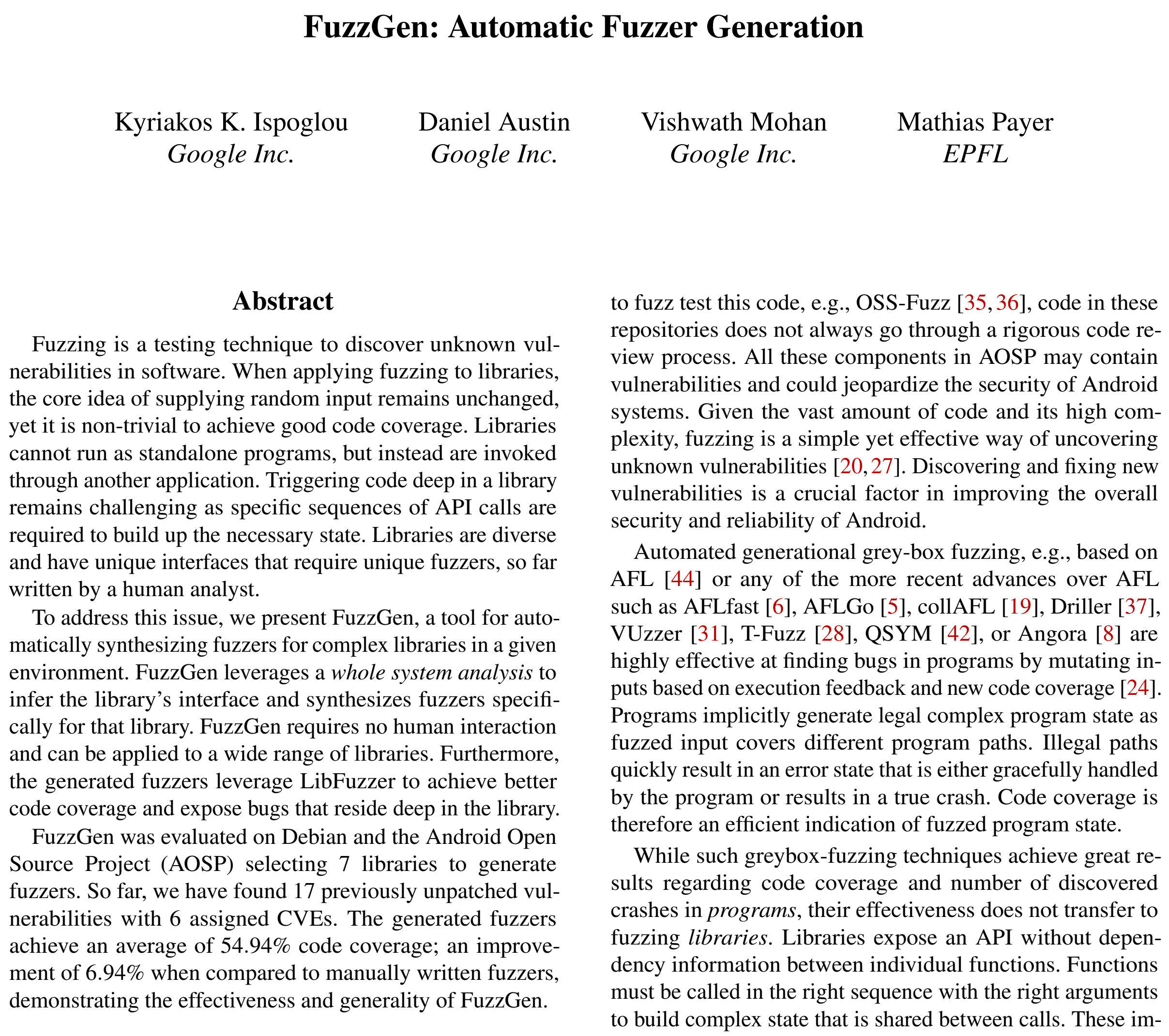
Fuzzing + code coverage /2

- The idea is to **progressively refine** (narrow down) the **space of possible inputs**, by focusing on inputs that **cause code coverage to increase / new code paths to be explored**
- For example, if a certain value for an input parameter causes the execution to terminate early, **there is no point** in fuzzing w/ that parameter set to that value
  - … even if there are other parameters that can be varied
  - Consider the previous example: if is_int = True, any of the infinite possible combinations of values for a, b **will not uncover the bug!**

Example – AFL++

- Older but popular fuzzer
- **Fuzzing approach:**
  - Record inputs that resulted in **exploring unique and/or previously unexplored code paths**
  - Prioritize picking those input those fuzzing cycle
  - Apply various **mutations** to selected inputs (bit flips, byte substitutions, etc.)
  - Use **genetic algorithms** to discover more/better test cases

Let’s talk about the paper



FuzzGen: Automatic Fuzzer Generation

Kyriakos K. Ispoglou, *Google Inc.*
Daniel Austin, *Google Inc.*
Vishwath Mohan, *Google Inc.*
Mathias Payer, *EPFL*

Abstract

Fuzzing is a testing technique to discover unknown vulnerabilities in software. When applying fuzzing to libraries, the core idea of supplying random input remains unchanged, yet it is non-trivial to achieve good code coverage. Libraries cannot run as standalone programs, but instead are invoked through another application. Triggering code deep in a library remains challenging as specific sequences of API calls are required to build up the necessary state. Libraries are diverse and have unique interfaces that require unique fuzzers, so far written by a human analyst.

To address this issue, we present FuzzGen, a tool for automatically synthesizing fuzzers for complex libraries in a given environment. FuzzGen leverages a *whole system analysis* to infer the library’s interface and synthesizes fuzzers specifically for that library. FuzzGen requires no human interaction and can be applied to a wide range of libraries. Furthermore, the generated fuzzers leverage LibFuzzer to achieve better code coverage and expose bugs that reside deep in the library.

FuzzGen was evaluated on Debian and the Android Open Source Project (AOSP) selecting 7 libraries to generate fuzzers. So far, we have found 17 previously unpatched vulnerabilities with 6 assigned CVEs. The generated fuzzers achieve an average of 54.94% code coverage; an improvement of 6.94% when compared to manually written fuzzers, demonstrating the effectiveness and generality of FuzzGen.

to fuzz test this code, e.g., OSS-Fuzz [35,36], code in these repositories does not always go through a rigorous code review process. All these components in AOSP may contain vulnerabilities and could jeopardize the security of Android systems. Given the vast amount of code and its high complexity, fuzzing is a simple yet effective way of uncovering unknown vulnerabilities [20,27]. Discovering and fixing new vulnerabilities is a crucial factor in improving the overall security and reliability of Android.

Automated generational grey-box fuzzing, e.g., based on AFL [44] or any of the more recent advances over AFL such as AFLfast [6], AFLGo [5], collAFL [19], Driller [37], VUzzer [31], T-Fuzz [28], QSYM [42], or Angora [8] are highly effective at finding bugs in programs by mutating inputs based on execution feedback and new code coverage [24]. Programs implicitly generate legal complex program state as fuzzed input covers different program paths. Illegal paths quickly result in an error state that is either gracefully handled by the program or results in a true crash. Code coverage is therefore an efficient indication of fuzzed program state.

While such greybox-fuzzing techniques achieve great results regarding code coverage and number of discovered crashes in *programs*, their effectiveness does not transfer to fuzzing *libraries*. Libraries expose an API without dependency information between individual functions. Functions must be called in the right sequence with the right arguments to build complex state that is shared between calls. These im-

Paper goals/methodology

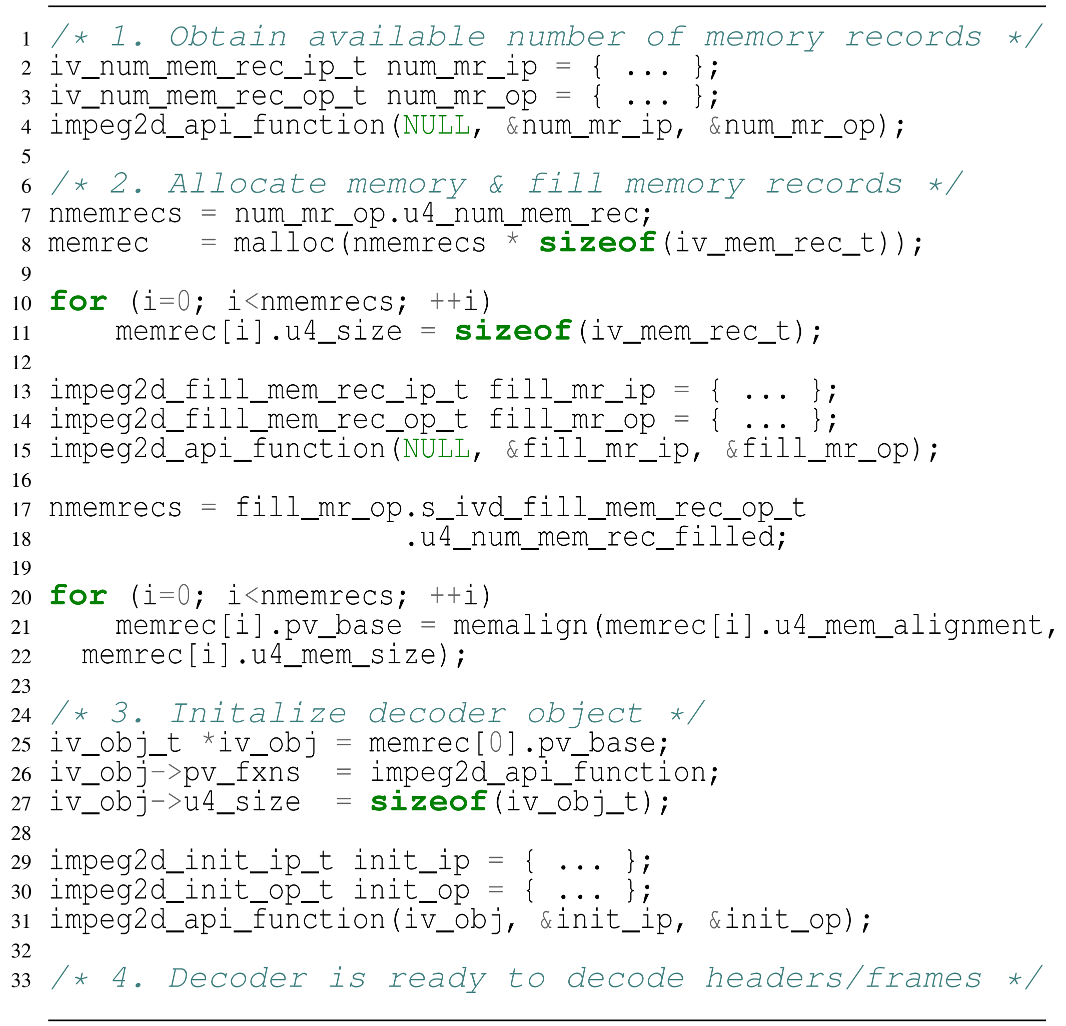
- Enable **fuzzing** of **libraries** (not just executables)
  - **Important** but **overlooked** target!
- **Methodology:** use **static analysis** to **extract information** about the libraries, use this information to **build fuzzing harness** for library

Why focus on libraries

- Traditional fuzzers are **fairly limited in scope**
- Fuzzing is achieved by **repeatedly executing a program** while **varying** either:
  - Standard input (input stream to the program)
  - Input files
- **Clearly does not work with libraries!**

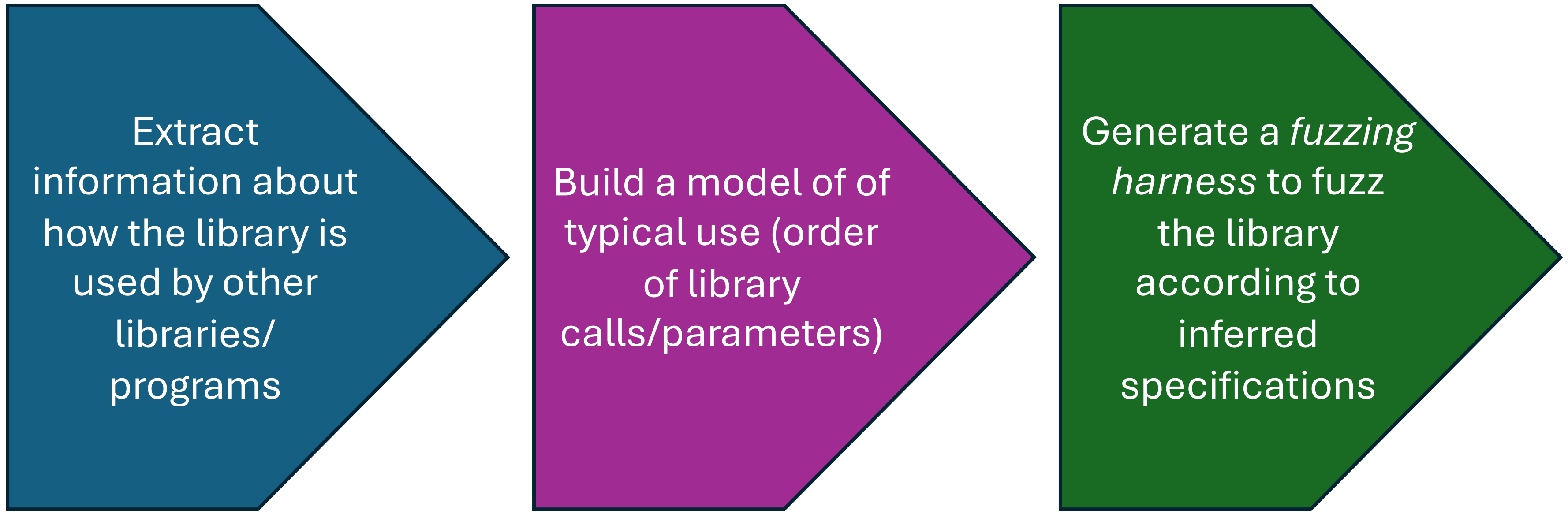
The problem with libraries

- Libraries differ from executables!
- Most importantly:
    - A library does not have a **single entry point** (unlike “main” or similar)
    - A library typically includes **multiple functions** that must be called in a **specific order**



```
/* 1. Obtain available number of memory records */
iv_num_mem_rec_ip_t num_mr_ip = { ... };
iv_num_mem_rec_op_t num_mr_op = { ... };
impeg2d_api_function(NULL, &num_mr_ip, &num_mr_op);

/* 2. Allocate memory & fill memory records */
nmemrecs = num_mr_op.u4_num_mem_rec;
memrec   = malloc(nmemrecs * sizeof(iv_mem_rec_t));

for (i=0; i<nmemrecs; ++i)
    memrec[i].u4_size = sizeof(iv_mem_rec_t);

impeg2d_fill_mem_rec_ip_t fill_mr_ip = { ... };
impeg2d_fill_mem_rec_op_t fill_mr_op = { ... };
impeg2d_api_function(NULL, &fill_mr_ip, &fill_mr_op);

nmemrecs = fill_mr_op.s_ivd_fill_mem_rec_op_t
                     .u4_num_mem_rec_filled;

for (i=0; i<nmemrecs; ++i)
    memrec[i].pv_base = memalign(memrec[i].u4_mem_alignment,
  memrec[i].u4_mem_size);

/* 3. Initalize decoder object */
iv_obj_t *iv_obj = memrec[0].pv_base;
iv_obj->pv_fxns   = impeg2d_api_function;
iv_obj->u4_size   = sizeof(iv_obj_t);

impeg2d_init_ip_t init_ip = { ... };
impeg2d_init_op_t init_op = { ... };
impeg2d_api_function(iv_obj, &init_ip, &init_op);

/* 4. Decoder is ready to decode headers/frames */
```

General technical approach



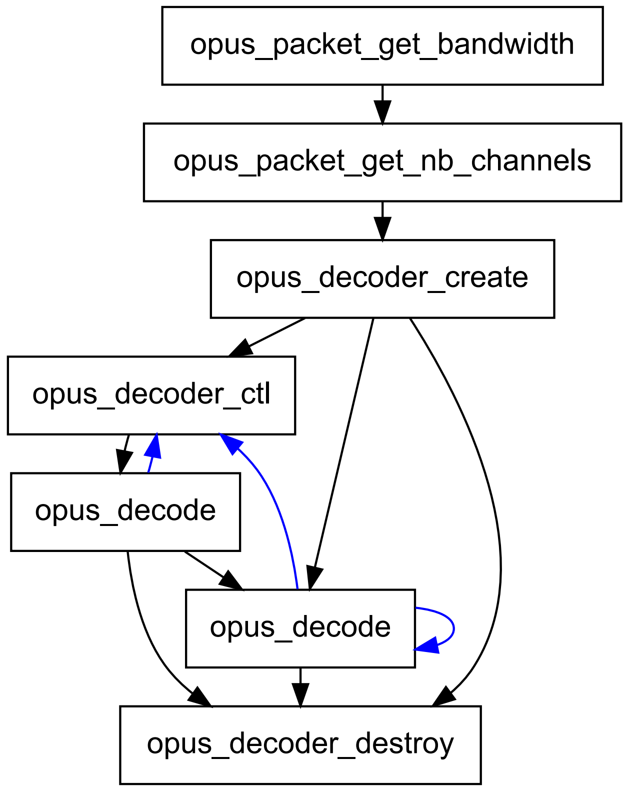
Extract information about how the library is used by other libraries/ programs

Build a model of of typical use (order of library calls/parameters)

Generate a *fuzzing harness* to fuzz the library according to inferred specifications

Analysis of uses

- **General idea:** scan a system (e.g. a Linux installation) for programs & libraries using the target library
- For each library user, use **static analysis** to build a **graph** depicting **how the library is used**
- Graph captures **temporal relationships** (interpreted as **control dependencies**) between calls, **parameters values** etc.

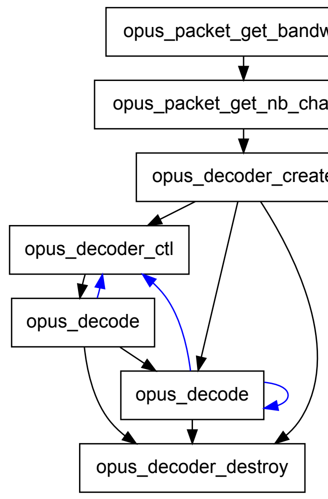


(b)

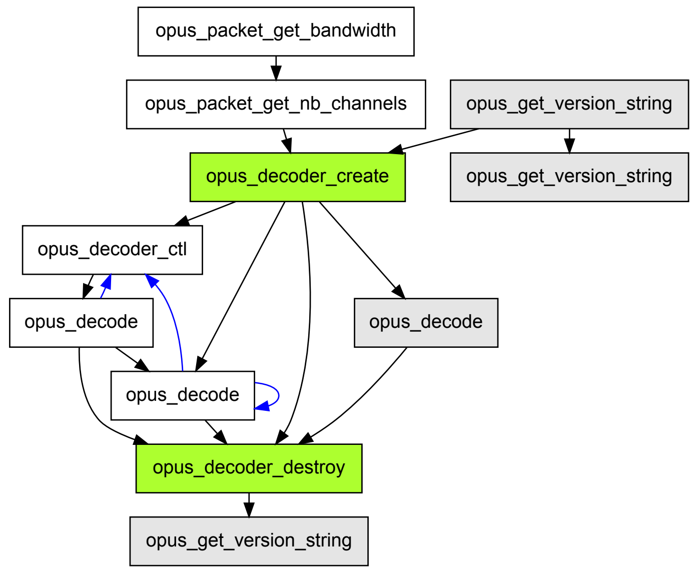
From graphs to models

- Rather than use extracted graphs as models for fuzzing, the approach attempt to **merge multiple graphs into one** to build **more general models of graph usage**
- General idea: find identical nodes in different graphs, copy all subtrees rooted in such nodes from one graph into the other
- When possible, **merge identical nodes**

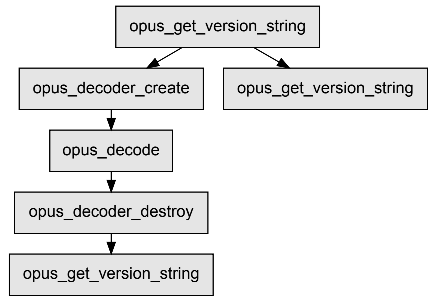
Example (from paper)



(b)



(d)



(c)

#1: opus_packet_get_bandwidth, opus_get_version_string
#2: opus_packet_get_nb_channels, opus_get_version_string
#3: opus_decoder_create
#4: opus_decoder_ctl, opus_decoder_decode
#5: opus_decoder_decode
#6: opus_decoder_decode
#7: opus_decoder_destory
#8: opus_get_version_string

(e)

What happens next?

- The graph is encoded in a **fuzzing stub**
- The fuzzing stub is a **program** which **incorporates the graph-based model** built previously
- **Different inputs** will result in **different paths through the graph** being traversed (and thus, hopefully, different modes of use for the library)
- The fuzzer does not know anything about all of this, but can observed that manipulating the input causes **different execution paths through the library** to be explored

Results – Code Coverage

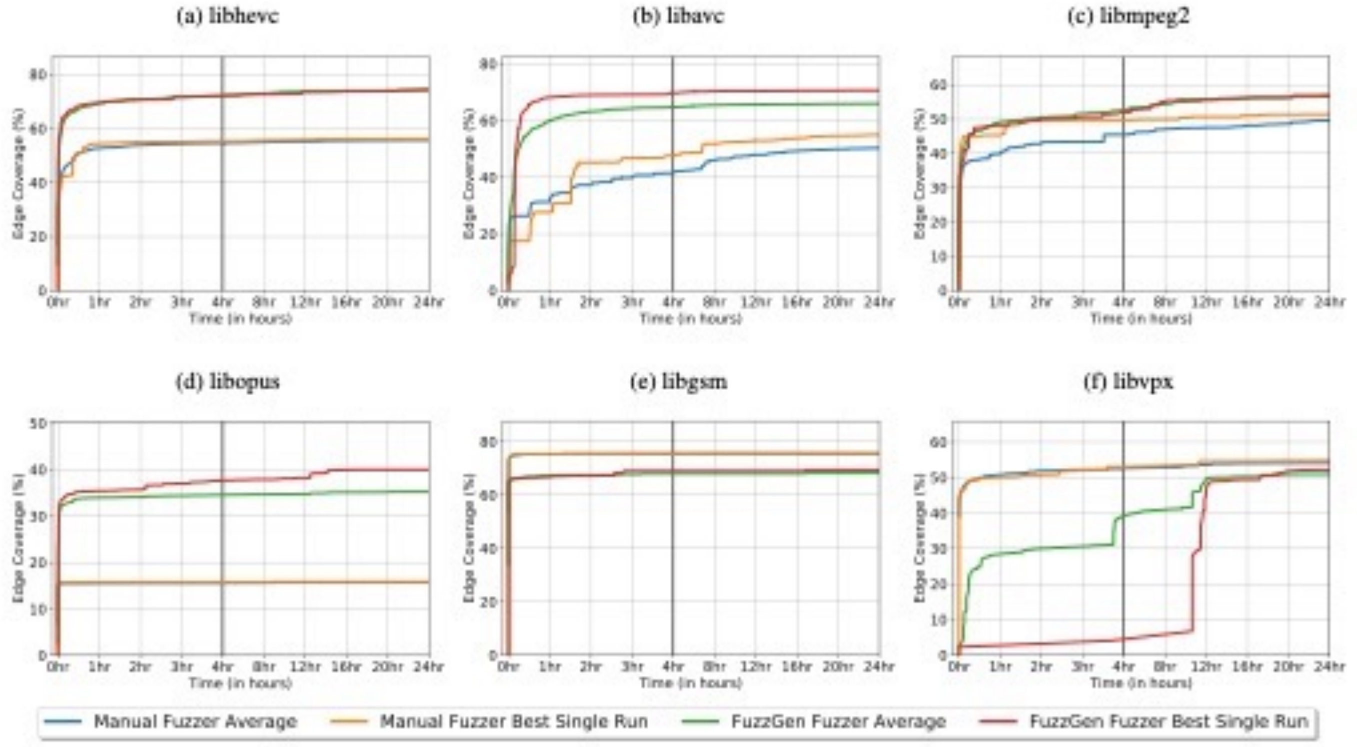


Figure 5: Code coverage (%) over time for each library. Blue line shows the average edge coverage over time for manual fuzzers and orange line shows the edge coverage for the best single run (among the 5) for manual fuzzers. Similarly, the green line shows the average edge coverage for FuzzGen fuzzers, and the red line the edge coverage from best single run for FuzzGen fuzzers.

Why is code coverage a good metric to evaluate a fuzzer?

More on fuzzing

- Fuzzing remains a **fairly active** area of research
- While fuzzing it isn’t a magic solution to hidden vulnerabilities/bugs, it is better than nothing and it is:
  - **Simple to implement**
  - **Simple to run** (e.g. in CI/CD pipelines)
- Other areas of research:
  - Coming up with better **fuzzing strategies** (is code coverage the best metric?)
  - Fuzzing programs which receive **more complex inputs** (e.g., server applications receiving input over the network)

See you next time!